CS 3710 Introduction to Cybersecurity
Term: Fall 2025

Lab Exercise 4 – Exploiting

Due Date: October 3, 2025 by 11:59pm ET
Points Possible: 7

Name: Laxmi Ghanate

By submitting this assignment you are digitally signing the honor code, "On my honor, I pledge that I have neither given nor received help on this assignment."

AI assistance is permitted on this assignment. Please be aware that AI answers are not always correct, so validate the answer. If you are opposed to using AI, you do not need to do so on the questions specifically requesting AI. Please cite all sources including AI.

1. Overview

As an ethical hacker you are scanning the target network and identify a potentially vulnerable server. You do some research and find a vulnerability and exploit for the target system. You then launch the exploit to gain root level access to the target!

2. Initial Setup

From your Virginia Cyber Range course, select the **Cyber Basics** environment. Click "Start My Environment" to start your environment and once it is ready click "Join My Environment" to open your Linux desktop.

Task 1: Perform a network scan to identify a potentially vulnerable server

In Lab 2 you used Nmap to scan your network to identify live targets and the ports open on each target. Review your previous results or complete a new network scan to identify a vulnerable target running Microsoft Directory Services also known as SMB or "Samba". You can also run the nmap scan again to identify the target if needed.

Question 1: What is your vulnerable target's IP address? (.5 point)

172.16.27.133

```
┌──(student㉿kali.example.com)-[~]
└─$ sudo nmap -sS -sV -p 139,445 --open -Pn -T4 172.16.16.0/20
Starting Nmap 7.95 ( https://nmap.org ) at 2025-09-28 19:42 UTC
Nmap scan report for ip-172-16-27-133.ec2.internal (172.16.27.133)
Host is up (0.00061s latency).

PORT    STATE SERVICE     VERSION
139/tcp open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: MYGROUP)
445/tcp open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: MYGROUP)
MAC Address: 12:AA:BE:62:27:7B (Unknown)
Service Info: Host: SMB
```

Question 2: What is the **specific** (X.Y.Z) version of the Samba service is running on your target? (.5 point)

```
PORT    STATE SERVICE     VERSION
139/tcp open  netbios-ssn Samba smbd 3.X - 4.X (workgroup: MYGROUP)
445/tcp open  netbios-ssn Samba smbd 4.6.0 (workgroup: MYGROUP)
MAC Address: 12:AA:BE:62:27:7B (Unknown)
Service Info: Host: SMB
```

Samba 4.6.0

Task 2: Examine the details of the vulnerability

You have done some research on these open services and versions and it looks like the best vulnerability to use for an exploit is going to be the Samba vulnerability CVE-2017-7494. Learn about this vulnerability at the National Vulnerability Database here:

https://nvd.nist.gov/vuln/detail/CVE-2017-7494

Search the Exploit database https://www.exploit-db.com/ to find a Metasploit module for the identified CVE number.

Question 3: What is the name of the Metasploit Module? (.5 point)

Samba 3.5.>0 < 4.4.14/4.5.10/4.6.4 - 'is_known_pipename()' Arbitrary Module Load (Metasploit)

Now that we have identified a vulnerability to exploit and know the Metasploit module name, it is time to get serious.

Task 3: Run Metasploit

Metasploit is a penetration testing framework that comes installed in Kali Linux. Metasploit commands are run from the command line.

First you need to start **Metasploit Framework Console (msfconsole)**. There are several steps to properly starting the **msfconsole**.

First, you need to start the postgresql database service. This database is used by Metasploit to store information gathered via penetration testing activities. You will have to provide the password (which is **student**) when running this command.

```
service postgresql start
```

Second, you will have to initialize the msf database using the **msfdb init** command as follows. You will need to use the **sudo** command to run this command with root level privileges.

```
sudo msfdb init
```

Finally, you can start the **Metasploit Framework Console** by using the **msfconsole** command as follows:

```
msfconsole
```

The msfconsole will start and give you the **msf>** prompt once the startup has completed. While you are in the msfconsole, regular Linux commands will no longer work.

To see a list of commands that are available from the **msf>** prompt, type a **?** and press enter. (you will need to scroll up to see everything)

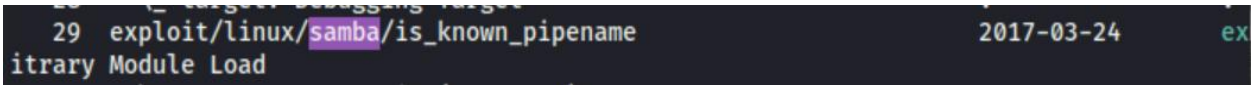
The first command you will use is the **search** command which will allow you to look for information on the Metasploit exploit that you will use for this penetration test.

You can search for a CVE number or a Metasploit module name. Use the **search** command to look for the Metasploit module that corresponds to the vulnerability you discovered.

The search command shows there is an exploit, the location of the exploit, disclosure date, the rank, and the description of the exploit. You can now use this information to exploit the target.

Question 4: What is the disclosure date and rank of the exploit? (.5 point)

Disclosure Date 2017-03-24 and Rank excellent



Next you will use the **use** command to load the exploit. When using the **use** command, you have to use the full path as shown in the name column of the search results.

The prompt will change to show the name of the exploit that was loaded.
Now use the **options** command to see the options for the exploit:

```
options
```

If you look at the **options** list, the first option **RHOST** is blank and is required. **RHOST** stands for **Remote Host** and is the IP address of the target system. Whenever you are attempting to exploit a target system, you always have to provide an **RHOST**. RPORT is also required but it is already set.

You can use the **set** command to set the **RHOST** option using the following command. Remember the **target_ip** is the IP address of the target system identified in Question 1.

```
set rhost target_ip
```

Once the **RHOST** option is set, you can then use the **exploit** command to launch the exploit.

If the exploit fails the first time, check to make sure the target IP address (**RHOST**) is correct using the **options** command and run the exploit again. If the exploit succeeds, you will get **Command shell session 1 opened** message. This means you have successfully executed the exploit against the target system.

After the **Command shell session 1 opened** message, you will just have a blinking cursor and no indication that you have entered a shell on the target system. Use the **whoami** command to see what account you are logged in as in the shell on the target system as follows:

```
whoami
```

Question 5: Paste a screenshot that shows your whoami here. (.5 point)

By the answer to whoami, you should know whether the exploit was successful. If so - Congratulations, if everything went well you now pwn the target system! You identified a target, identified a vulnerability in that target, and used Metasploit to exploit the target to get root shell access to the target.

At this point you can run other commands such as **pwd**, **ls**, etc. to learn about your exploited target system.

The basic shell is a little difficult to work with as it gives you no prompt and no feedback if the command you execute fails. You can get a more usable shell by using a python script. Use the following command to create a more useful shell on the target system:

```
python -c 'import pty; pty.spawn("/bin/bash")'
```

This command uses the Python programming language to create a new bash shell. Bash is the default shell used in Linux.

Now that you pwn the system let's grab a copy of the /etc/shadow file. An attacker would copy this file offline to crack user passwords and try the same passwords on other systems. Now you get to apply what you have learned in Lab 1 and crack the passwords in the compromised /etc/shadow file.

Question 6: Paste a screenshot that shows the cracked passwords from target's /etc/shadow file here. You might not crack all of them, but if you get more than a dozen, you are good to go. (1 point)

Task 4: Identifying and Correcting Potential Buffer Overflows

Buffer overflow attacks are often a direct result of poor programming practices. Examine the following code and answer the questions below it (both examples were covered in class):

```
void main()
{
        char source[] = “username12”;
        char destination[8];
        strcpy(destination, source);

        return 0;
}
```

Question 7: Explain why this code has the potential for a buffer overflow. Make sure to mention the function and the weakness of the function. (.5 point)

This code can overflow because source ("username12") is 11 bytes (including the NUL) while destination is only 8 bytes, so strcpy(destination, source) writes past the end of destination. strcpy is unsafe because it copies until the NUL terminator with no bounds checking. To fix it, either allocate a destination large enough (e.g., char destination[sizeof source];) or use a bounded copy like snprintf(destination, sizeof destination, "%s", source) (or strlcpy/strncpy with careful NUL termination).

Question 8: Show how you can fix this code to mitigate the buffer overflow (AI may not give the correct answer, for example, the answer is not just making the size of the destination buffer larger). (1 point)
Fix A — use a bounded write (snprintf)

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char source[] = "username12";
   char destination[8];

   /* snprintf writes at most sizeof(destination)-1 characters and always
      NUL-terminates (unless size == 0). This prevents overflow. */
   snprintf(destination, sizeof destination, "%s", source);

   return 0;
}
```

Examine the following code and answer the questions below it:

```
#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
    {
        printf ("\n Wrong Password \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass)
    {
       /* Now Give root or admin rights to user*/
        printf ("\n Root privileges given to the user \n");
    }

    return 0;
}
```

Question 9: Explain why this code has the potential for a buffer overflow. Make sure to mention the function and the weakness of the function. (.5 point)

This code is vulnerable because it uses gets(buff), which reads input until a newline without any bounds checking. If an attacker supplies more than 15 bytes, the extra bytes will be written past buff on the stack and can overwrite nearby variables (like pass) or control data, allowing privilege escalation or arbitrary code execution. gets is inherently unsafe, always use a bounded input function (e.g., fgets(buff, sizeof buff, stdin)) or otherwise enforce a maximum length.

Question 10: Show how you can fix this code to mitigate the buffer overflow (again, the answer is not just making the size of the buffer larger). (1 point)

One way to fix the code is fgets() into a fixed buffer. This would mean fgets reads at most sizeof buff - 1 characters and always NUL-terminates (when not NULL), preventing overflow. We also strip the newline before comparing

```
#include <stdio.h>
#include <string.h>

int main(void)
{
```

```
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    if (fgets(buff, sizeof buff, stdin) == NULL) return 1;

    /* remove trailing newline if present */
    buff[strcspn(buff, "\n")] = '\0';

    if (strcmp(buff, "thegeekstuff") != 0) {
        printf("\n Wrong Password \n");
    } else {
        printf("\n Correct Password \n");
        pass = 1;
    }

    if (pass) {
        printf("\n Root privileges given to the user \n");
    }
    return 0;
}
```

Question 11: What is the difference between a heap-based and stack-based buffer overflow? Give a real world example for each. (.5 point)

Stack-based overflow: Happens in a function's stack frame when input exceeds the buffer and overwrites control data like the saved return address often leading to code execution. A Real-world example is the 1988 Morris worm exploited a gets()-based stack overflow in the fingerd daemon to gain remote shell access.

Heap-based overflow: Happens in dynamically allocated memory. Overwriting adjacent heap objects or allocator metadata can hijack control flow (e.g., corrupting function pointers/vtables) or enable arbitrary writes. Real-world example is MS04-028 (Windows GDI+ JPEG parsing) was a heap overflow triggered by crafted image data, allowing code execution when a malicious JPEG was processed.

By submitting this assignment you are digitally signing the honor code, "I pledge that I have neither given nor received help on this assignment".

END OF EXERCISE

References

https://metasploit.help.rapid7.com/docs